Timemory: Modular Performance Analysis for HPC

Jonathan R. Madsen, Ph.D.
Application Performance Group

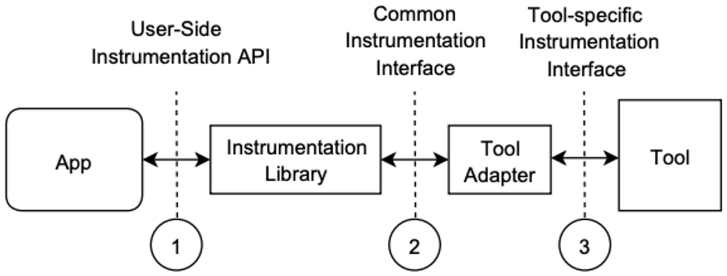
Aug 1[illegible], 2020

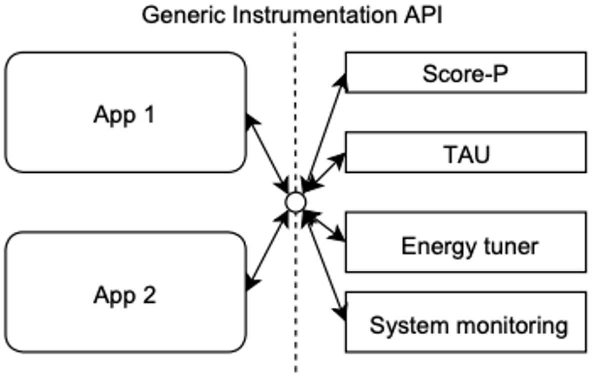
Key Takeaways

1. Not “use X *or* timemory” → “use X *via* timemory” (“…and arbitrarily add support for Y”)
2. Front-end library: optional, arbitrary to implement
3. Back-end library: *highly modular and easily extendable*
4. Tools are provided but is, first and foremost, a *toolkit*

Objective: Common Instrumentation API

- Abstractions to record performance data for a section of code or work-flow are a staple in HPC apps
- Issue:
  - Increase in multiple language apps
  - Increase in multiple architectures apps
- Result:
  - Comprehensive analysis requires using multiple tools and/or multiple APIs
  - Supporting new language/architectures in abstractions tends to require significant re-write





Boehme, D, Huck, K A, Madsen, J, and Weidendorfer, J. Thu . "The Case for a Common Instrumentation Interface for HPC Codes". United States. https://www.osti.gov/servlets/purl/1574633.

Common Instrumentation Goals

- All common HPC languages
- Instrumentation and sampling
- Data-sharing between tools
- Local customization
- Optimal efficiency and optimal flexibility implementations
- Agnostic to input application
- Agnostic to output by tools

- Minimal overhead
- Extensible by application
  - Wrap existing abstractions
- Easy to support
  - Stable interface
  - In-house modifications do not require upstream propagation

Common Instrumentation Requirement

- Common convention for calling tools
  1. Function pointers? Callbacks?
     - Restricts invocation to specific function signature(s)
     - Collecting phase data can be unwieldy
  2. Dynamic Polymorphism?
     - Restricts invocation to specific function signatures(s)
     - Simplifies phase data collection → member data
     - Always requires heap allocation

Common Instrumentation Problem

- Need methodology which abstracts types out of pattern(s) for calling tools while preserving types
  - Start/Stop + different input types:
    - Host timer requires no input
    - Stream-specific CUDA event timer requires cudaStream_t input
  - Get + different return types:
    - Trip-counter → returns integer
    - Hardware counter → returns array of doubles
    - Roofline → returns elapsed time paired with hardware counters

Common Instrumentation Solution

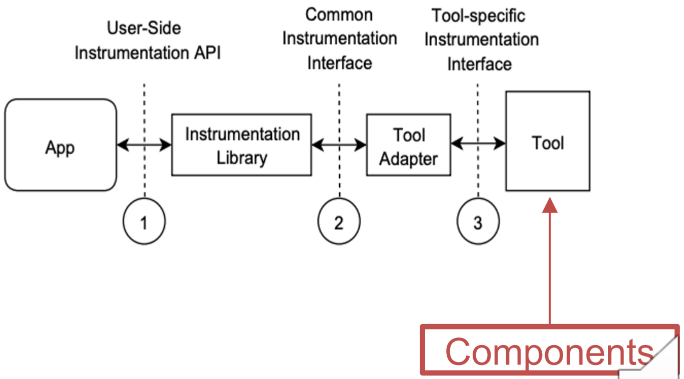
- Each tool wraps its behavior/requirements into struct(s)
  - Static configuration data, local intermediate (phase) data, etc.
- Calling convention is member function *names*
  - Functions are overloaded for supported input types
  - Functions return any type
- Variadic template classes
  - Bind tools together into tool-like struct
  - Variadic member functions accept any input + deduce return type
  - Manage metadata, other behavior (e.g. data-sharing)

Overview

component	C++ structs which encapsulate a measurement instance
operation<T>	C++ structs which encapsulate how to perform the measurement in different scenarios (different arguments, cached data available, etc.)
bundle<T…>	C++ struct which binds together components and maps components onto operations + broadcasts arguments to operations
storage<T>	C++ struct which manages accumulation of persistent data and call-stack hierarchy
trait<T, …>	C++ structs which encode properties at compile-time
settings	C++ struct which maps configuration to code accessors, environment variables, command-line arguments, etc.

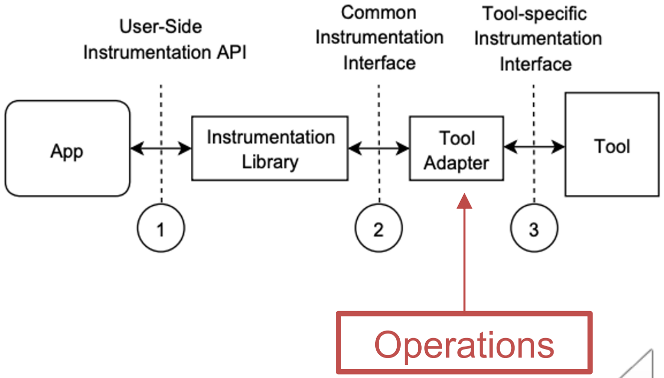
Components

- Components provide "tools"
  - Member functions + overloads + type-traits → define capabilities
  - Reusable within other components (ideally)
  - Could implement other components
  - Could provide aforementioned:
    - Callback system
    - Dynamic polymorphism
  - May be defined by external tool



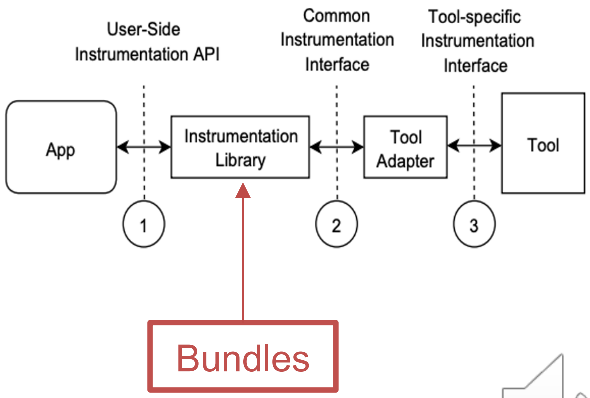
Operations

- Operations provide “tool adapter”
  - Handle member function invocation for given set of arguments
    - Using SFINAE and/or tag dispatch
  - Local customization via template specialization
  - Usually require component instance
  - May provide function call operator
  - Handle runtime disabling



Bundles

- Bundles are the “instrumentation library”
  - Zero or more components are template parameters
  - Filter out components marked as “unavailable”
    - Important for creating a portable handle to tools
- Bundles come in various flavors
  - Implicit start/stop via RAII
  - Implicit/explicit interaction with storage
  - Different component allocation schemes
    - Heap, stack, mixed



Putting It All Together

Component Interface ⇨

```
struct Foo
{
    void start();
};
```

```
struct Bar
{
    void start();
    void start(int);
};
```

```
struct Yaz
{
    void sample();
};
```

bundle<T…> ⇨

```
bundle<Foo, Bar, Yaz> obj;
obj.start(10);
```

⇩ Variadic template unrolling

operation<T> ⇨

```
operation::start<Foo>(Foo& f, int v);
operation::start<Bar>(Bar& b, int v);
operation::start<Yaz>(Yaz& y, int v);
```

⇩ SFINAE

```
f.start();
b.start(10);
```

Summary

- Flexible interface for creating handling multiple tools without abstraction
- Ideal for built-in performance analysis into your code
  - Automatically support multiple tools
  - Fully customize interface
  - Developers *and users* can create custom components

Acknowledgement

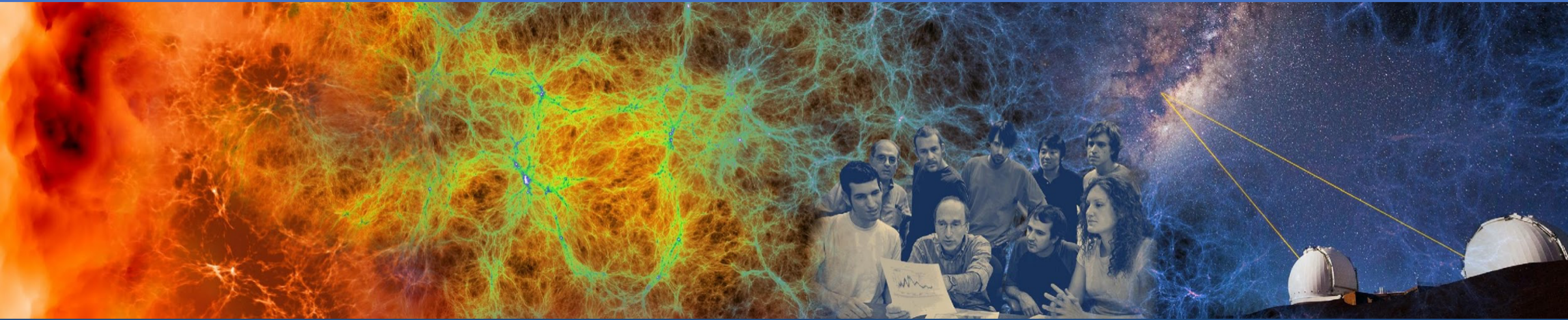
Authors from Lawrence Berkeley National Laboratory were supported by the U.S. Department of Energy's Advanced Scientific Computing Research Program under contract DE-AC02-05CH11231.

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-05CH11231.

Additional Information

- Back-up slides contain more information on:
  - Data storage model, type-traits, settings
  - Instrumentation in (more) detail
  - Wrapping common instrumentation around external functions
  - Python bindings
  - Summary of existing pre-built tools
  - Various other utilities
  - Future Work
  - Relevant Links
  - Contact Information

Backup Slides



Common Performance Analysis Paradigms

- Start / stop collecting metric around region
- Sample metrics at regular intervals
- Record event data within application
- Dynamic instrumentation
- Compiler instrumentation
- Accumulate or separate data per-process, per-thread
- Call-graph hierarchy

Performance Analysis APIs

1. Tools provide APIs
   - VTune → ittnotify
   - NVIDIA → NVTX
   - Caliper, CrayPAT, gperftools, LIKWID, Score-P, TAU, etc.
2. Libraries provide tool APIs
   - MPI → PMPI, MPI-T, OpenMP → OMPT, Kokkos → KokkosP
3. Libraries provide built-in implementations of tool APIs
   - AMReX has amrex::TinyProfiler (basic timing), CrayPAT, ARM-Forge, NVTX, VTune, etc.

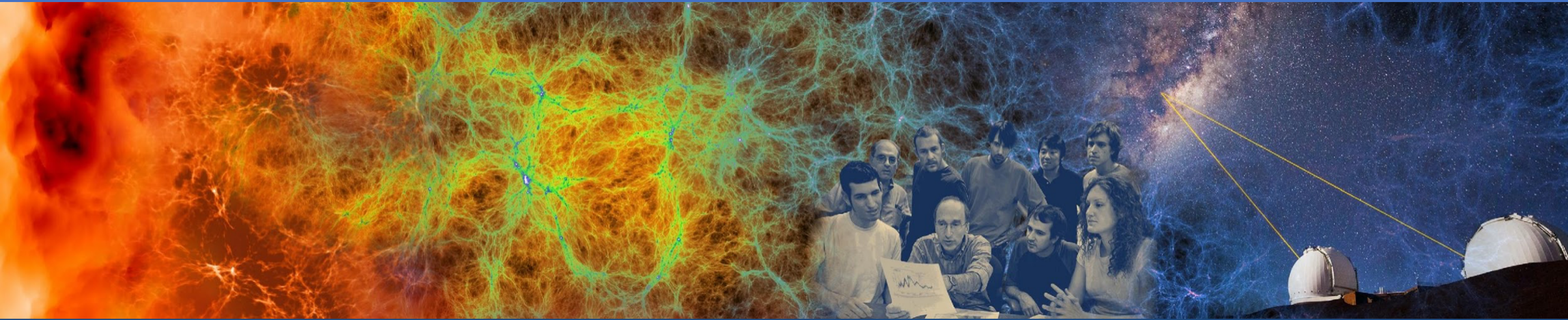
Common instrumentation simplifies #3 by generalizing #1 and providing implementations for #2

Proposal

- Use C++ templates to provide skeleton framework
  - Performance analysis tools use common set of patterns
- Use C++ template metaprogramming to:
  - Unify dissimilar interfaces
  - Accommodate different data storage requirements
  - Minimize abstractions and/or opaque data types (i.e. void*)
  - Eliminate any runtime logic available at compile-time
- Use C++ variadic templates to:
  - Support creating a single type which controls multiple tools
  - Provide functions which can receive any inputs

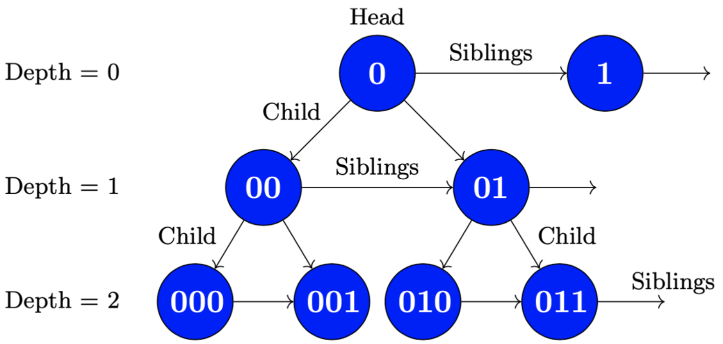
Timemory

- Storage
- Traits and Settings
- Example



Storage

- Singleton *per component* which handles tracking the persistent data
  - Unique to each thread and/or process
  - No synchronization locks or communication overhead outside of construction and destruction
  - Secondary instances get merged into primary during finalization
- Supports intermixed layouts: tree, flat, timeline, tree-timeline, flat-timeline
- Writes to various output formats
  - JSON, XML, text, binary, flamegraph
  - More can be supported



Call-graph per component. Each node is keyed to a label (e.g., function name, file, and line number) and contains an instance of the component. The component instance within the call-graph provides data-storage only.

Traits and Settings

- Traits customize behavior, optimize performance, add/remove features
  - {start,stop}_priority, uses_{timing,memory}_units, statistics, derivation_types, cache, python_args, uses_storage, etc.
- Settings handle mapping runtime configuration variables to environment variables, command-line arguments
  - tim::settings::enabled() [C++]
  - timemory.settings.enabled [Python]
  - TIMEMORY_ENABLED [environment variable]
  - --timemory-enabled=off [command-line argument]

Example

```
TIMEMORY_DECLARE_COMPONENT(Foo)
TIMEMORY_DEFINE_TRAIT(flat_storage, component::Foo, true_type)
TIMEMORY_DEFINE_TRAIT(uses_timing_units, component::Foo, true_type)

TIMEMORY_DECLARE_COMPONENT(Bar)
TIMEMORY_DEFINE_TRAIT(start_priority, component::Bar,
                      priority_constant<1>)
TIMEMORY_DEFINE_TRAIT(stop_priority, component::Bar,
                      priority_constant<-1>)

TIMEMORY_DECLARE_COMPONENT(Yaz)
TIMEMORY_DEFINE_TRAIT(is_available, component::Yaz, false_type)

TIMEMORY_DECLARE_COMPONENT(Egg)
TIMEMORY_DEFINE_TRAIT(uses_storage, component::Egg, false_type)
```

- Four components: Foo, Bar, Yaz, Egg
- Foo always implements flat storage
- Bar has start/stop priority
- Yaz is unavailable
- Egg does not use storage

```
namespace tim { namespace component {
struct Foo : base<Foo, int64_t>
{
    void start();
    void stop();
    double get() const;
};

struct Bar : base<Foo, std::vector<double>>
{
    void start(cudaStream_t);
    void stop(cudaStream_t);
    std::vector<double> get() const;
};

struct Egg : base<Egg, void>
{
    void start(int);
    void stop();
};
}}
```

Usage

```cpp
using bundle_t = tim::component_tuple<
    Foo, Bar, Yaz, Egg>;

void spam(cudaStream_t stream)
{
    bundle_t obj("spam");
    obj.start(stream);
    // ...
    obj.stop(stream);
    auto ret = obj.get();
}
```

- Yaz is implicitly removed from *bundle_t*
- Foo, Bar, and Egg created on stack
- Only Foo and Bar insert into storage
- Foo is started w/o args
- Bar is started with stream arg
- Egg does not provide compatible start
- Bar has priority → stops first
- Foo, Egg stop according to template order
- Foo, Bar add themselves to storage node
- obj.get() translates to tuple of types w/ non-void get() member function

Implementation

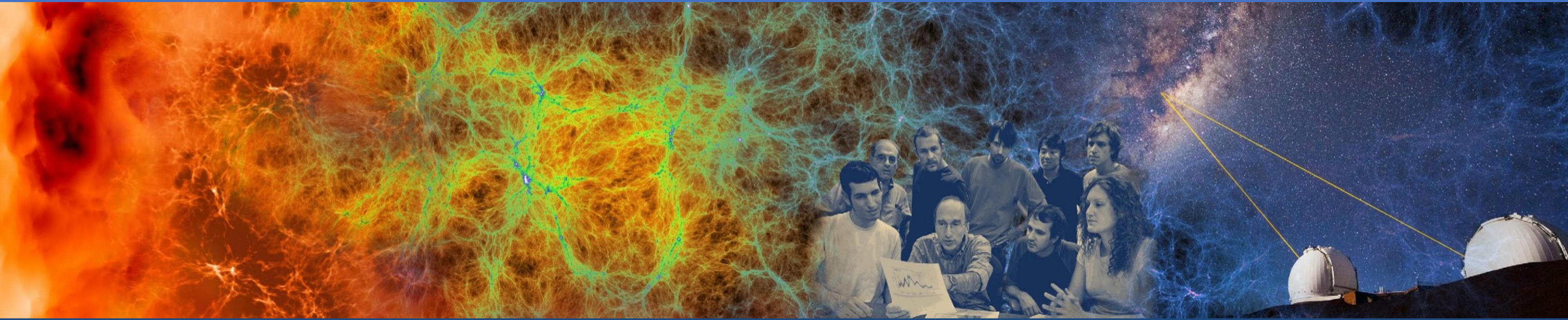
```cpp
using bundle_t = tim::component_tuple<
    Foo, Bar, Egg>;

void spam(cudaStream_t stream)
{
    auto _hash = get_hash("spam");
    Foo f;
    Bar b;
    Egg e;
    operation::insert_node(f, _hash, scope::flat{});
    operation::insert_node(b, _hash, scope::tree{});
    f.start();
    b.start(stream);
    // ...
    b.stop(stream);
    f.stop();
    e.stop();
    operation::pop_node{f};
    operation::pop_node{b};
    auto ret = std::make_tuple(
        f.get(), b.get());
}
```

via operation::start<T>

via operation::stop<T>

What else does timemory provide?



Instrumenting External Functions

- Simple function wrapping and replacement via GOTCHA
  - Wrap N functions in ~N+3 lines of code
    - Templates will automatically extract return-type and arguments
    - Macro for unmangled (extern “C”) functions
    - Macros for mangled (extern “C++”) free-, member-functions
  - <u>Supports auditing arguments and return value</u>
  - Reference counting → scoped function wrapping/replacement

```
struct mpip  {};
using toolset_t       = component_tuple<component::wall_clock>;
using mpip_gotcha_t   = component::gotcha<246, toolset_t, mpip>;
```

```
TIMEMORY_C_GOTCHA(mpip_gotcha_t, 79, MPI_Gather);
TIMEMORY_C_GOTCHA(mpip_gotcha_t, 80, MPI_Gatherv);
TIMEMORY_C_GOTCHA(mpip_gotcha_t, 81, MPI_Get);
TIMEMORY_C_GOTCHA(mpip_gotcha_t, 82, MPI_Get_accumulate);
TIMEMORY_C_GOTCHA(mpip_gotcha_t, 83, MPI_Get_address);
TIMEMORY_C_GOTCHA(mpip_gotcha_t, 84, MPI_Get_count);
TIMEMORY_C_GOTCHA(mpip_gotcha_t, 85, MPI_Get_elements);
TIMEMORY_C_GOTCHA(mpip_gotcha_t, 86, MPI_Get_elements_x);
TIMEMORY_C_GOTCHA(mpip_gotcha_t, 87, MPI_Get_library_version);
TIMEMORY_C_GOTCHA(mpip_gotcha_t, 88, MPI_Get_processor_name);
TIMEMORY_C_GOTCHA(mpip_gotcha_t, 89, MPI_Get_version);
TIMEMORY_C_GOTCHA(mpip_gotcha_t, 90, MPI_Graph_create);
```

Tools

- Numerous pre-built command-line tools are provided
  - **timem** → combines UNIX time + rusage + HW counter
  - **timemory-run** → dynamic instrumentation via Dyninst
  - **timemory-avail** → provides component info, available settings, available HW counters
- Numerous pre-built instrumentation libraries are provided
  - **timemory-mpip** → wraps MPI calls with runtime selection of components and tracks comm data sizes
  - **timemory-ncclp** → wraps NCCL calls with runtime selection of components and tracks comm data sizes
  - **timemory-ompt** → implement OMPT with runtime selection of components
  - **kokkosp_timemory*** → generic and dedicated implementations of the KokkosP interface

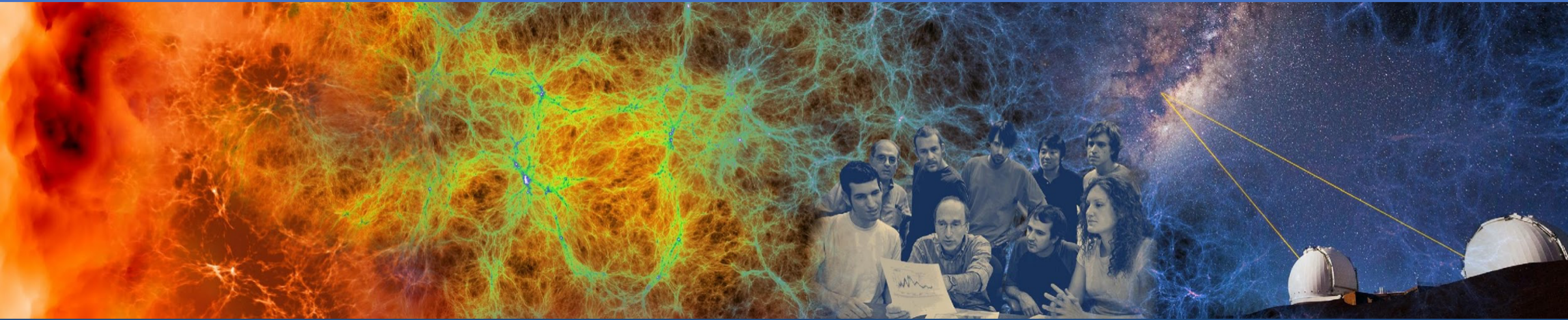
Python

- Extensive Python bindings are provided
  - Settings
  - Decorators, context-managers
  - Python profiling (per-function), tracing (per-line)
  - Control over instrumentation libraries
  - Python classes for bundles
  - Generic plotting, roofline plotting
- Every enumerated component gets standalone Python class, e.g.
  - timemory.component.PapiVector
  - timemory.component.CuptiActivity
  - timemory.component.CaliperMarker
- Common workflows for CI

Utilities

- CMake INTERFACE libraries for various flags, libs
- Empirical roofline toolkit (ERT)
- Argument parser similar to Python argparse
- Embedding Python interpreter
- Subprocess piping utilities
- Conditional instrumentation
- String concatenation
- Signal handling and sampling
- Generic source location class
- Platform-agnostic environment get/set templates
- STL container overloads for arithmetic and statistics

Summary



Summary (1/3)

- Toolkit for building performance analysis tools
  - Use timemory-provided tools and libraries to perform analysis with timemory-provided components
  - Use timemory API to implement custom high-level performance monitoring system
  - Use timemory API to supplement timemory-provided tools with custom components
  - Build custom tool or library with timemory as backend
  - Build custom Python tool with timemory standalone components
- Components
  - Optimal invocation of one measurement / usage

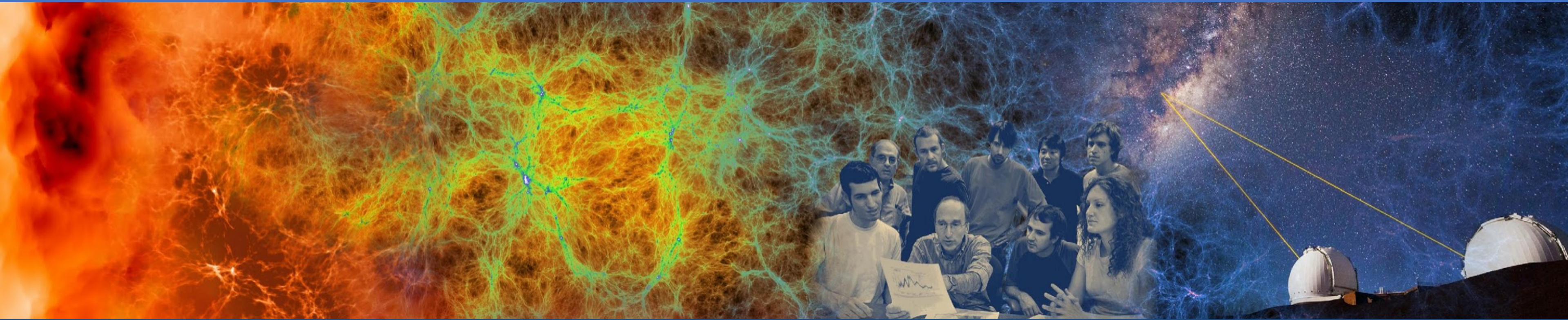
Summary (2/3)

- Operations provide:
  - Ability to specialize behavior for a given set of arguments
  - Use SFINAE to handle support for operation and/or arguments
- Bundles broadcast arguments to operations and provide common interface to components with different features/capabilities
- Storage is optional feature and simplifies aggregation over multiple processes and multiple threads
- Type-traits encode properties and features at compile-time
- Settings provide mapping between environment variables, code accessors, and command-line arguments

Summary (3/3)

- Powerful, easy-to-use GOTCHA extensions
- Numerous command-line tools and instrumentation libraries are provided
- Python interface provides:
  - Timemory runtime
  - Toolkit of components for building Python tools
  - Visualization utilities
- Miscellaneous utilities for common performance analysis tool requirements

- Future Work
- Relevant Links
- Contact Info



Future Work

- Planned
  - Compiler Instrumentation
  - Score-P, CUPTI Profiler API, OpenCL, ROCprofiler, ROCtracer
  - Additional serialization formats (e.g. YAML, CUBE, etc.)
  - Jupyter notebooks
  - NERSC Iris + SLURM Integration
  - Components which perform analysis of other components
    - Produce recommendations/hints w.r.t. bottlenecks
- Potential
  - LLVM X-ray Support
  - Generic LLVM pragma implementation

Relevant Links

- Journal Article
  - Madsen, J.R. et al. (2020) Timemory: Modular Performance Analysis for HPC. In: Sadayappan P., Chamberlain B., Juckeland G., Ltaief H. (eds) High Performance Computing. ISC High Performance 2020. Lecture Notes in Computer Science, vol 12151. Springer, Cham
- Source code
  - github.com/NERSC/timemory
- Documentation
  - timemory.readthedocs.io
- Tutorials
  - github.com/NERSC/timemory-tutorials
- Package Managers
  - Available via Spack, PyPi
  - github.com/NERSC/timemory/wiki/Installation-Examples

Contact Info

- If you are interested in:
  - Integrating timemory into your performance analysis workflow
  - Contributing a component for your tool
  - Hands-on tutorial
  - Feature request

  please send an email to jrmadsen@lbl.gov or create an issue on GitHub